# *Conceito Gerais:*

- → Definição de Sistema Operativo;
  - → Para que são otimizados:
    - Em servidores;
    - Em *desktop/laptop*;
    - Em *tablets/smartphones*;
    - Em sistemas embutidos;
- → Breve história do sistema UNIX;
  - → Caracteríticas dos sistemas UNIX;
  - → Sistemas mais comuns e a sua linhagem:
    - LINUX;
    - MAC OS X;
    - Windows;
    - iOS;
    - Android;
- → Funções básicas de um SO;
- → Arquitetura:
  - *Kernel*;
  - Serviços típicos;
  - *System Calls*;
- → Multi-programação e Multi-tarefa;
- → BIOS:
  - O que é;
  - Para que serve;
- → Bootstrap no UNIX;
- → O que são *interrupts* e *traps* (exemplos);
- → Interação com dispositivos:
  - Controladores dos dispositivos;
  - *Device drivers*;
  - DMA (para dispositivos com elevada largura de banda);
- → Proteção do sistema:
  - Modos *kernel* e *user*;
  - *System Calls* (como *traps*);
  - Implementação;

# *UNIX*

O Unix é um sistema operativo criado no início dos anos 70, principalmente por *Dennis Ritchie* e *Ken Thompson*. As suas principais características técnicas são sua portabilidade, sua capacidade de multi-utilizador e de multitarefa, eficiência, alta segurança e o bom desempenho em tarefas de rede. Existem vários sistemas operativos conhecidos como derivados do Unix, entre eles, o ***Linux***, ***MacOS***, ***Android*** e o ***iOS*** (derivado do ***Mac OS X***).

***Linux***, foi criando no início dos anos 90 por *Linus Torvalds* quando o núcleo desenhado por *Linus* foi distribuído juntamente com as ferramentas ***GNU***. O projeto GNU criou a licença de software GPL, que garante as liberdades de uso, modificação e colaboração a respeito do software. Desta forma, as ferramentas do GNU são desenvolvidos, revistas, melhoradas e adaptadas para milhares de utilizadores e centenas de empresas ao redor do mundo. Além disso, este conjunto está disponível gratuitamente e seu código de fonte é aberto.

***MacOS*** foi desenvolvido pela *Apple Inc.* A sua versão clássica foi desenvolvida entre 1984 até 2001 e depois criou-se o modelo mais desenvolvido que ainda é usado hoje em dia. É o segundo sistema operativo mais usado atualmente, estando atrás do ***Microsoft Windows***, desenvolvido pela *Microsoft* em 1985.

**Android** é um sistema operativo móvel baseado em uma versão modificada do *kernel Linux* e outro software de código aberto, projetado principalmente para *touchscreen,* desenvolvido por um conjunto de firmas chamado de *Open Handset Alliance* em 2007 e comercializado pela *Google*.

i***OS as*** é um sistema operativo móvel criado e desenvolvido pela *Apple Inc* e lançado em 2007, exclusivamente para o seu *hardware*. É o segundo sistema operativo móvel mais instalado, depois do *Android*.

# *Sistemas Operativos*

**Definição do livro:** Um sistema operativo é o único programa em execução o tempo todo no computador – geralmente chamado de *kernel*.

**Definição da aula:** Um sistema operativo é um *software* que providencia os recursos de *hardware* de um computador necessários à execução das aplicações dos seus utilizadores.

- Atua como intermediário entre o utilizador e o *hardware*
- Providencia o básico para as aplicações
- Alguns são desenhados para tornar o sistema do computador conveniente e de fácil utilização, outros são desenhados para usar o *hardware* com maior eficiência e outros para combinar os dois.

Os sistemas operativos são desenhados para corresponder da melhor forma aos dispositivos em que estão.

- **Servidores**: Projetado para maximizar a utilização de recursos e garantir que todo o tempo de CPU, memória e I/O disponíveis sejam usados de maneira eficiente e justa entre todos os utilizadores -***"Keep all users happy!"***
- ***Desktop/Laptop:*** Otimizados para a experiência de um único utilizador em vez dos requisitos de vários utilizadores. Projetado para facilitar a utilização com alguma atenção ao desempenho e quase nenhuma para a utilização dos recursos;
- ***Tablets/Smartphone:*** Otimizados para usabilidade e minimização do gasto da bateria. Projetado para facilidade de uso, com atenção especial à utilização dos recursos;

- ***Embedded Computers:*** A maioria dos computadores embutidos, em dispositivos domésticos e automóveis, têm pouca ou nenhuma interface para o utilizador. Projetado para ser executado sem/mínima intervenção do utilizador;

Um sistema de um computador pode ser dividido em quatro partes:

- ***Hardware*** – recursos básicos do sistema (CPU, memória, dispositivos I/O…);
- **Sistema operativo** – controla e coordena o uso do *hardware* com as várias aplicações e utilizadores;
- **Programas –** definem a maneira como os recursos do sistema são usados para resolver as necessidades dos usuários (processadores de texto, navegadores *web*, sistemas de base de dados, jogos, compiladores, ...)
- **Utilizadores –** pessoas, outros programas/computadores;

O sistema operativo moderno geralmente inclui os seguintes componentes principais:

1) Gestão de **processos**
   - Criar, suspender, retomar e encerrar processos (usuário/sistema);
   - Fornecer mecanismos para comunicação de processos;
   - Fornecer mecanismos para sincronização de processos;
   - Fornecer mecanismo para tratamento de impasses (*deadlock)*
2) Gestão de **memória**;
   - Alocação e desalocação de espaço de memória conforme necessário;
   - Rastreamento de quais partes da memória estão a ser usadas no momento e por quem;
   - Decidir quais processos/dados devem ser movidos para dentro e para fora da memória e quando;
3) Gestão de **armazenamento**;
   - Fornecer visão uniforme e lógica do armazenamento de informações;
   - Suporte para criar, excluir e manipular ficheiros e diretórios;
   - Políticas de controlo de acesso para determinar quem pode acessar o quê;
   - Mapeamento/*Backup* de ficheiros em dispositivos de armazenamento secundário não volátil;
4) Gestão de dispositivos de **I/O**;
   - Oculta peculiaridades dos dispositivos de *hardware* do usuário;
   - Responsável pela gestão de memória de I/O;

O ***kernel*** é o centro essencial de um sistema operativo de computador. É o núcleo que fornece serviços básicos para todas as outras partes do sistema operativo. É a camada principal entre o sistema operativo e o *hardware* e ajuda na gestão de processos e memória, sistemas de ficheiros, controle de dispositivos e rede.

Os serviços prestados diferem de um sistema operativo para outro, mas podemos identificar classes comuns:

- Interfaces de usuário – para permitir operação e controle eficazes do sistema **(*User Interface*)**;
- Execução do programa – para carregar um programa na memória e executá-lo **(*Program Execution*)**;
- Operações de I/O – para fornecer um meio de realizar operações de I/O **(*I/O Operations*)**;

- Sistemas de ficheiros – para permitir a manipulação eficaz de ficheiros e diretórios **(*File Systems*)**;
- Comunicações – para permitir a troca de informações entre processos no mesmo computador ou entre computadores em uma rede **(*Communications*)**;
- Deteção de erros – estar constantemente ciente de possíveis erros que podem ocorrer no *hardware* da CPU/memória, nos dispositivos de I/O ou nos programas do usuário, a fim de tomar as medidas apropriadas para garantir uma computação correta e consistente **(*Error Detection*)**;

Outro conjunto de serviços existe não para ajudar o utilizador, mas sim para garantindo o funcionamento eficiente do próprio sistema:

- Alocação de recursos – quando vários processos estão sendo executados simultaneamente, os recursos disponíveis (como ciclos de CPU, memória principal, armazenamento de ficheiros, dispositivos de I/O) devem ser alocados de forma eficiente para cada um deles **(*Resource allocation*)**;
- Contabilidade – para acompanhar quais utilizadores usam quanto e quais tipos de recursos do computador **(*Accounting*)**;
- Proteção e segurança – para evitar que processos simultâneos interfiram uns com os outros ou com o próprio sistema operativo e para proteger o sistema de terceiros **(*Protection and Security*)**;

As ***system calls*** fornecem uma interface para os serviços do sistema operativo. São acessadas a partir de uma *high-level application program interface (API)*. Podem ser agrupadas em seis categorias:

- Controlo de processo (*Process Control);*
- Manipulação de ficheiros (*File Manipulation*);
- Manipulação de dispositivos (*Device Manipulation*);
- Manutenção de informações (*Information Maintenance*);
- Comunicação (*Communication*);
- Proteção (*Protection*);

Um dos aspetos mais importantes dos sistemas operacionais é a capacidade de ter **vários programas em execução.** A **multiprogramação** aumenta a utilização da CPU organizando os processos para que a CPU possa sempre executar um trabalho.

(1) O sistema operativo começa a executar um processo por meio do agendamento de processos (*job scheduling)*;
(2) Eventualmente, o trabalho pode ter que esperar por alguma tarefa, como uma operação de I/O;
(3) Em um sistema não multiprogramado, a CPU ficaria inativa;
(4) Em um sistema multiprogramado, o sistema operativo alterna para outro trabalho. Quando esse trabalho precisa esperar, a CPU muda novamente para outro trabalho e assim por diante. Eventualmente, o primeiro trabalho termina de esperar e recupera a CPU;
(5) Enquanto pelo menos um trabalho precisa ser executado, a CPU nunca fica inativa;

A **multiprogramação** aumenta a utilização da CPU, mas não fornece necessariamente a interação do utilizador com o sistema do computador. ***Multitasking*** (multi-tarefa) é uma extensão da **multiprogramação** que aumenta o tempo de resposta em que a CPU alterna tarefas com tanta frequência que os utilizadores podem interagir com cada tarefa enquanto ela está em execução.

**BIOS (Basic Input/Output System)** é o primeiro programa responsável pela inicialização do sistema do computador (*bootstrap program)*, é carregada na inicialização ou no *reboot*.

- Armazenada em memória de leitura apenas (ROM, *read-only memory*);
- Inicializa todos os aspetos do sistema, desde dos registros da CPU para controladores de dispositivos até conteúdos da memória;
- Carrega o *kernel* do sistema operativo e inicia a sua execução;

Assim que o *kernel* é carregado, pode começar a fornecer os serviços disponíveis aos utilizadores. Alguns serviços são fornecidos fora do *kernel*, por processos do sistema que são carregados no momento da inicialização (no UNIX, o primeiro processo do sistema é o processo *init* que inicia muitos outros processos do sistema). Assim que esta fase é concluída, o sistema é totalmente inicializado e começa a aguardar a ocorrência de algum evento.

A ocorrência de um evento geralmente é sinalizado por uma interrupção (***interrupt***) do *hardware* ou do *software.* O *hardware* pode acionar uma interrupção a qualquer momento enviando um sinal para a CPU, para comunicar que ele precisa da atenção do sistema operativo. O *software* pode acionar uma interrupção executando uma operação especial chamada de *System Call.*

Quando a CPU é interrompida, suspende a atividade corrente, guardando o estado em que está, e transfere , de imediato, a execução para uma função fixa chamada de *interrupt handler* (manipulador de interrupção) para lidar com o evento. Após a conclusão, a CPU retoma a computação do processo que tinha interrompido.

***Traps*** são levantadas pelo programa do utilizador para invocar uma funcionalidade do sistema operativo. Suponha que o programa do utilizador exija a impressão de algo na tela. Ele invocaria uma ***trap*** e o sistema operativo executaria essa instrução. São usadas principalmente para implementar ***system calls***. Um ***interrupt*** é gerado por um dispositivo de *hardware*, as interrupções são assíncronas, isto é, podem ocorrer a qualquer momento. Dispositivos como teclados são conectados ao processador através do pino de interrupção. Quando uma tecla é pressionada, ela gera uma interrupção. O processador mudará do processo atualmente em execução para um *interrupt handler*. Nesse cenário, o manipulador de interrupção do teclado é chamado. Depois de completar a rotina de tratamento de interrupção, o processador volta para o programa original que estava sendo executado.

Os sistemas do computador consistem em vários **controladores de dispositivos** (componentes de *hardware*) conectados por meio de um *bus – sistema de comunicação que transfere dados entre componentes dentro de um computador ou entre computadores*. Para se comunicar com cada controlador de dispositivo, os sistemas operativos exigem um ***driver* de dispositivo (*Device drivers*)** específico (componente de software). ***Device drivers*** ajudam o *kernel* a executar ações. São pedaços de código que correspondem a cada dispositivo e são executados quando os dispositivos se conectam ao sistema operativo ou ao *hardware*. Ajudam a fechar o espaço entre as aplicações e o *hardware.* Para garantir a funcionalidade correta, o *kernel* deve ter um ***device driver*** embutido para cada periférico presente no sistema.

Para iniciar uma operação de I/O, o **driver de dispositivo** carrega os registros apropriados no **controlador de dispositivo.**

(1) O controlador, por sua vez, examina o conteúdo desses registros para determinar qual ação tomar (por exemplo, ler um caractere do teclado).
(2) O controlador então inicia a transferência de dados do dispositivo para o *buffer* local.

(3) Uma vez que a operação de I/O foi concluída, o **controlador do dispositivo** informa o ***driver* do dispositivo** por meio de uma **interrupção.** O ***driver* de dispositivo** retorna o controlo ao sistema operativo.

Esta forma de I/O controlada por *interrupt* é boa para transmitir pequenas quantidades de dados, mas pode haver sobrecarga se os dados forem em massa. Para dispositivos de I/O de alta velocidade, capazes de transmitir informações em velocidades próximas às da memória, esse problema é resolvido usando o **acesso direto à memória (DMA – *Direct Memory Access*)**, exemplos, *graphics cards, network cards, disk drive controllers…* O **controlador de dispositivo** transfere blocos de dados de seu próprio armazenamento de *buffer* diretamente para a memória principal, sem intervenção do **driver de dispositivo** (CPU).

Um sistema operativo projetado adequadamente deve garantir que um programa incorreto ou mal-intencionado não possa fazer com que outros programas sejam executados incorretamente. A abordagem adotada pela maioria dos sistemas de computador é fornecer suporte de hardware que nos permita diferenciar entre, pelo menos, dois modos separados de operação:

- ***User Mode;***
- ***Kernel Mode;***

A operação de modo duplo permite que o sistema operativo proteja a si mesmo e a outros componentes do sistema.

Um ***mode bit*** vindo do *hardware* indica o modo atual:

- Permite distinguir quando o sistema está a correr em *user code* ou em *kernel code;*
- Algumas instruções, designadas como instruções privilegiadas, são executáveis apenas no modo kernel (instruções para controle de I/O, gestão do *timer*, gestão de interrupção, …);
- Interrupções ou *system calls* alteram o modo para o *kernel*, o retornar das interrupções ou *system calls* redefinem-no para o modo *user;*

Normalmente, um número é associado a cada *system call* e a *system call interface* mantém uma tabela indexada de acordo com esses números. A *system call interface* invoca a *system call* pretendida no *kernel* do sistema operativo e retorna o status da *system call* e quaisquer valores de retorno.

# *Programas e Processos*

→ Programa: ficheiro binário executável;
- → Estrutura;
- → Como é obtido por compilação;

→ Processo e espaço de endereçamento
- → Segmentos e suas funções:
  - → *.text*;
  - → *.data*;
  - → *.bss*;
  - → *Heap;*
  - → *Stack;*

→ *Process Control Block (PCB);*
- → O que é;
- → Que informação contém;

→ Ciclo de vida de um processo;

→ Processo "*scheduler*";

→ "*Context Switch*";

→ Criação, gestão e terminação de processos;

→ Processos órfãos e *zombies;*

→ *Inter Process Communication* (*IPC*);

→ *API* de memória partilhada do UNIX/*System V;*

→ *API POSIX* de memória partilhada (*mmap*);

→ *Pipes;*

→ *Sockets UNIX* e *INET*;

→ Modelo cliente-servidor;

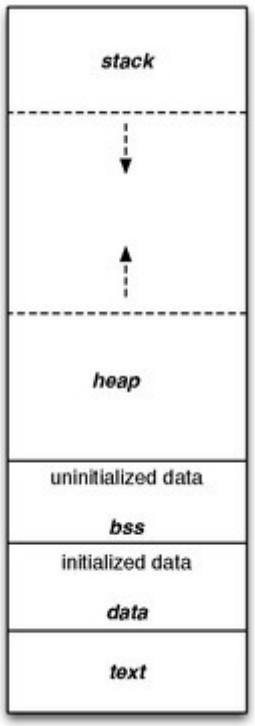
Cada ficheiro executável (**programa**) é composto por um *header*, descrevendo os segmentos de memória, seguido pelos dados do ficheiro. Os segmentos contêm informações necessárias para a execução do ficheiro durante a execução do mesmo, assim como, dados para vinculações (*linkers*) e de realocação. É produzido por compilação e ligação (*compiling and linking)* numa etapa.

Um **processo** é a unidade de trabalho na maioria dos sistemas e pode ser pensado como um programa em execução. Para cumprir o trabalho, um processo precisa de recursos, por exemplo, tempo de CPU, memória, ficheiros, etc.

Um **processo** possui várias partes:

- Secção de texto – contém o código do programa (***Text section***);
- Secção dos dados – contém as variáveis globais (***Data section***);
- *Heap* – contém memória dinamicamente alocada durante o tempo de execução do processo (***Heap***);
- Stack (Pilha) – contém dados temporários (como parâmetros de funções, endereços de retorno e variáveis locais) (***Stack***);
- Bloco de controlo do processo (PCB) – que inclui o contador do programa e os registradores da CPU (**Process Control Block**);

As partes de texto, dados, heap e pilha formam o espaço de endereço de memória do processo.




O sistema operativo representa cada processo por um ***Process Control Block (PCB)***, bloco de controlo de processo, que contém informações associadas ao processo:

- **Estado do processo** e **identificação do processo**;
- **Contador do programa**, a próxima instrução a ser executada;
- **Registradores da CPU**, **registradores de índice**, ***stack pointers***, etc;
- **Informações de agendamento**, como prioridades, *scheduling queue pointers,* etc;
- **Informações da memória**, a memória alocada para o processo;
- **Informações de contabilidade**, como CPU usada, tempo de relógio decorrido desde o início, limites de tempo, etc;
- **Informações de I/O**, como dispositivos de I/O alocados, lista de ficheiros abertos, etc.

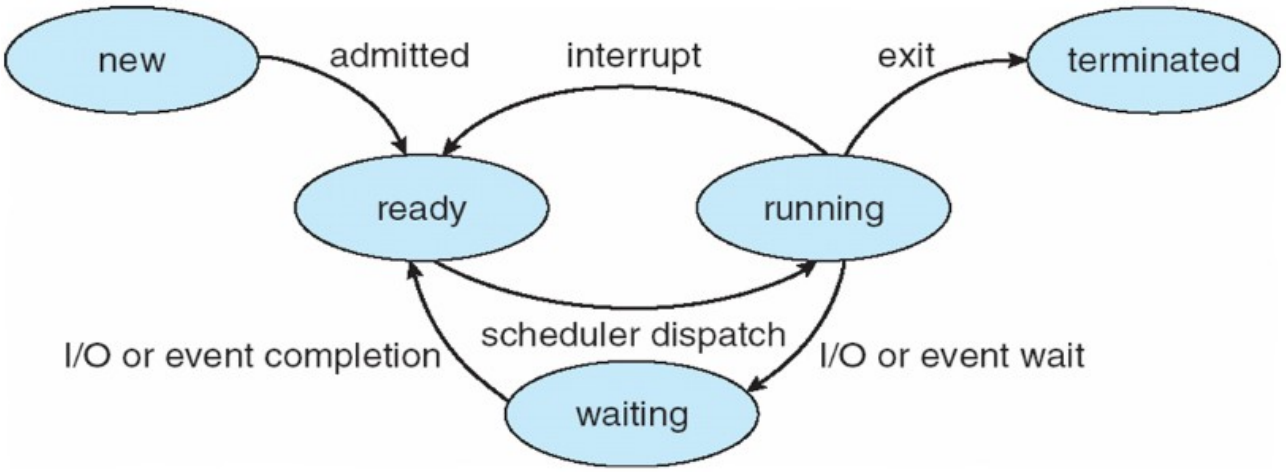
Um **programa** é uma entidade passiva (geralmente chamado de ficheiro executável) que contém uma lista bem definida de instruções (algoritmo). Um **processo** é uma entidade ativa correspondente a uma sequência de execução de um programa. Um **programa** torna-se num processo quando é carregado na memória.

- Um programa pode ser vários processos;
- Dois processos associados ao mesmo programa são considerados duas sequências de execução separadas;

À medida que um **processo** é executado, ele muda de estado de acordo com a sua atividade atual. Um **processo** pode estar em um dos seguintes estados:

- **Novo**, o processo está a ser criado;
- **Em execução**, as instruções estão a ser executadas;
- **Aguardando/Bloqueado**, o processo está a aguardar a ocorrência de algum evento;
- **Pronto**, o processo está a aguardar para ser atribuído a um processador;
- **Terminado**, o processo terminou a execução;

Muitos processos podem estar **prontos** ou **em espera,** mas apenas um processo pode ser executado em um processador de cada vez .




***Process scheduler*** é a atividade do gestor de processos que trata da remoção do processo em execução da CPU e da seleção de outro processo com base em uma determinada estratégia. É uma parte essencial de um sistema operativo de multiprogramação.

***Context Switch*** é a tarefa de alternar a CPU para outro processo.

- *Context Switch* requer salvar o contexto do processo atual no seu ***PCB***

O tempo do ***context switch*** é pura sobrecarga, porque o sistema não faz nenhum trabalho útil durante a troca. A velocidade de comutação depende da velocidade da memória, do número de registros que devem ser copiados e da existência de instruções especiais

- Quanto mais complexo for o SO e o PCB, maior será o ***context switch***

Durante a execução, um processo pode criar vários novos processos.

- O processo criador é chamado de ***parent process*** e os novos processos são chamados de filhos desse processo;
- Cada processo novo pode criar outros processos, formando uma árvore de processos;

A maioria dos sistemas operativos identifica os processos de acordo com um identificador de processo exclusivo (ou ***pid***), que normalmente é um número inteiro.

- O ***pid*** fornece um valor único para cada processo no sistema e pode ser usado como um índice para acessar vários atributos de um processo dentro do *kernel*

➢ Alternativas de compartilhamento de recursos:
  - Pais e filhos compartilham todos os recursos;
  - Os filhos compartilham um subconjunto dos recursos dos pais;
  - Pai e filho não compartilham recursos;

➢ Alternativas de execução:
  - Pai e filhos são executados simultaneamente;
  - O pai espera até que alguns ou todos os filhos tenham terminado;

➢ Alternativas de espaço de endereço:

- Pai e filho são duplicados (o filho começa com o programa e os dados do pai);
- O processo filho tem um novo programa carregado nele;

O que é preciso para criar um processo?

- Atribuir um identificador de processo e armazenar o novo PCB – Barato;
- Copiar o estado de I/O do pai (dispositivos de I/O alocados, lista de ficheiro abertos,…) – Meio caro;
- Configurar novas tabelas de memória para o espaço de endereço – Mais caro;
- Copiar dados do processo pai – Muito caro / Mais barato com copy-on-write;

No **UNIX/Linux,** um processo é criado pelo método *fork()* do *system call.*

- O novo processo (filho) consiste em um cópia do espaço de endereço do processo original (pai);
- O código de retorno para *fork()* é zero para o filho e o identificador do processo (***pid***) do filho é retornado para o pai (diferente de zero);
- Ambos os processos continuam a execução concorrentemente na instrução após o *fork()*;
- Esse mecanismo permite que o processo pai se comunique facilmente com os seus filhos;

Normalmente, após um *fork()*, um dos dois processos (pai ou filho) invoca a *system call exec()* – substitui o espaço de memória do processo (texto, dados, *heap*, partes da *stack*) por um novo programa do disco e começa a executar o novo programa na sua função principal, destruindo a imagem do processo anterior.

- No entanto, como nenhum processo novo é criado, o processo de chamada mantém o seu contexto (PCB);
- Dessa forma, pais e filhos podem seguir caminhos separados;

Um processo encerra a execução quando invoca explicitamente a *system call exit()* ou quando executa a última instrução.

- Todos os recursos do processo – incluindo memória física e virtual, ficheiros abertos e *buffers* de I/O – são então desalocados pelo sistema operativo;
- Um valor de *status* de saída (número inteiro) é disponibilizado para o pai por meio da *system call wait()*;

Um processo também pode encerrar a execução por meio do seu pai:

- A criança excedeu os recursos alocados;
- A tarefa atribuída à criança não é mais necessária;
- O pai está saindo e o sistema operativo não permite que o filho continue se o pai terminar (terminação em cascata – *Cascading Termination);*

Se um processo foi encerrado e nenhum pai está em espera, o processo encerrado é conhecido como **processo *zombie***.

- Somente quando o pai chama *wait()*, o ***pid*** do **processo *zombie*** e a sua entrada na tabela de processos são libertados;

Se um processo pai termina antes dos seus filhos, esses filhos, se houver, são conhecidos como **processos órfãos.**

- UNIX/Linux endereça **processos órfãos** atribuindo o processo ***init*** como o novo pai para os **processos órfãos**;
- O processo ***init*** invoca periodicamente *wait()*, permitindo assim que o identificador do processo do órfão e a entrada da tabela de processos sejam libertados.

Os processos dentro de um sistema podem ser **independentes** ou **cooperativos:**

- **Processo independente** não pode afetar ou ser afetado pela execução de outros;
- **Processo cooperativo** pode afetar ou ser afetado pela execução de outros;

Principais razões para **processos cooperativos**:

- **Partilha de informações** – acesso simultâneo à mesma informação;
- **Modularidade** – quebra a computação em subtarefas que fazem mais sentido;
- **Aceleração** – executa subtarefas simultâneas em paralelo;

Para trocar dados e informações, os **processos cooperativos** precisam de suporte para mecanismos de comunicação entre processos (***Interprocess communication – IPC***). Há dois modelos IPC:

- Passagem de mensagens (***Message passing***);
  - Comunicação via envio/receção de mensagens;
  - Funciona em rede;
- Memória partilhada (***Sharing memory***);
  - A comunicação ocorre simplesmente lendo/escrevendo na memória partilhada;
  - Pode levar a problemas complexos de sincronização;

Memória partilhada é mais rápida que a passagem de mensagens

- A passagem de mensagens é implementada usando *system calls* e por isso requerem uma tarefa mais demorada de intervenção do *kernel;*
- Com a memória partilhada, *system calls* só são necessárias para estabelecer as regiões de memória partilhada e, após isso, todos os acessos são tratados como acessos de memória de rotina e com nenhuma intervenção do *kernel*.

***Pipes( | )*** foram um dos primeiros mecanismo ***IPC*** nos primeiros sistemas UNIX e fornecem uma das maneiras mais simples para os processos se comunicarem. Permitem a comunicação em um estilo padrão *producer-consumer:*

- O produtor grava em uma extremidade do canal;
- O consumidor lê do outro lado;

São unidirecionais, permitindo apenas comunicação unidirecional, caso a comunicação bidirecional seja necessárias, dois *pipes* devem ser usados, com cada *pipe* enviando dados em uma direção diferente.

No UNIX/Linux, *pipes* são criadas com o comando *pipe()* do *system call*. O UNIX/Linux trata as *pipes* como um tipo especial de ficheiro, o que permite serem acessadas usando *read()* e *write()* do *system call*.

***Pipes*** não podem ser acessadas de fora do processo onde foram criadas.

- Normalmente, um processo pai cria um ***pipe*** e o usa para se comunicar com um processo filho criado por meio de *fork()*;
- Como um ***pipe*** é um tipo especial de ficheiro, o filho herda o ***pipe*** do processo pai;
- Depois do *fork*, podemos decidir em que direção flui a informação.

***Sockets*** são definidos como um terminal para a comunicação. Um par de comunicação entre processos através de uma rede usa um par de ***sockets*** – um para cada processo. Um ***socket*** é identificado por um endereço IP concatenado com um número. Em geral, os ***socket*** usam uma arquitetura **cliente-servidor**. O servidor aguarda os pedidos recebidos do cliente ouvindo uma porta

especificada. Uma vez que um pedido é recebido, o servidor aceita uma conexão do ***socket*** do cliente para completar a conexão.

***IPC Socket*** é um terminal de comunicação de dados para troca de dados entre processos em execução no mesmo sistema operativo *host*. ***INET Socket*** no Linux são ***sockets*** de *Internet,* ou seja, ***sockets*** baseados no protocolo IP.